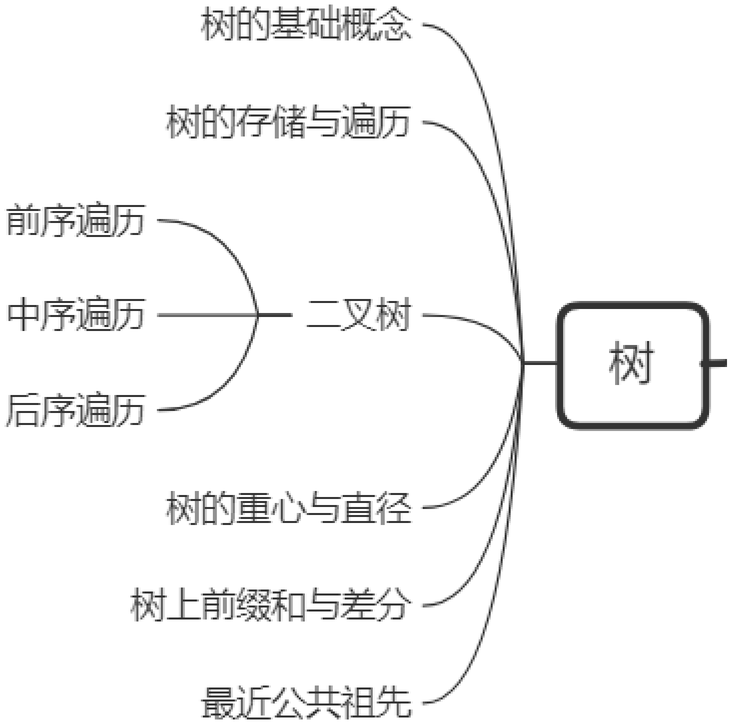
蓝桥杯十天冲刺省一

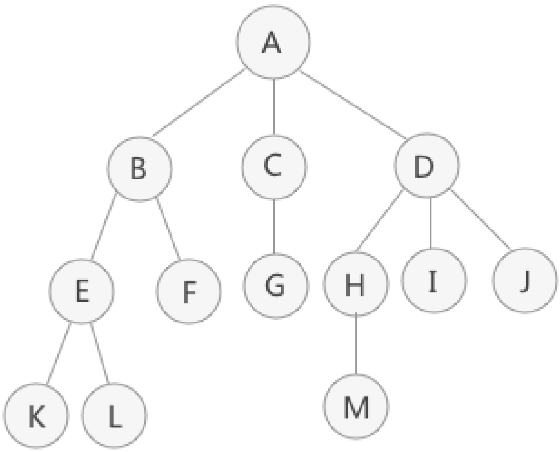

树的基础概念
树的存储与遍历
前序遍历
中序遍历
二叉树
后序遍历
树
树的重心与直径
树上前缀和与差分
最近公共祖先

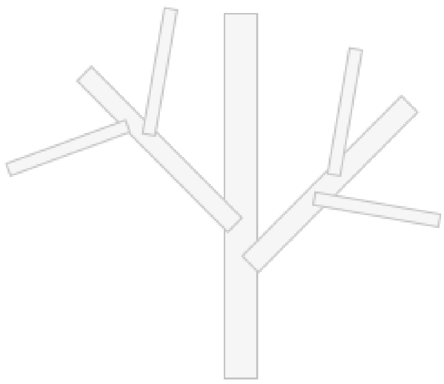
Day-7 树

之前学习了数组、字符串、队列、栈等等数据类型和数据结构，它们都是线性存储结构。本章要学习的树结构是一种非线性存储结构，存储的是具有“一对多”关系的数据元素的集合。

树结构不论是在竞赛中，还是在实际的工程开发中，都是一类重要的非线性数据结构，树中的节点之间具有明确的层次关系，并且每个节点会“分支”出若干个其他节点。

数据结构中的树和现实生活中的树长得一样，只不过我们习惯于处理问题的时候把树根放到上方来考虑。这种数据结构看起来像是一个倒挂的树，因此得名。后面我们提到的树均指数据结构中的树。





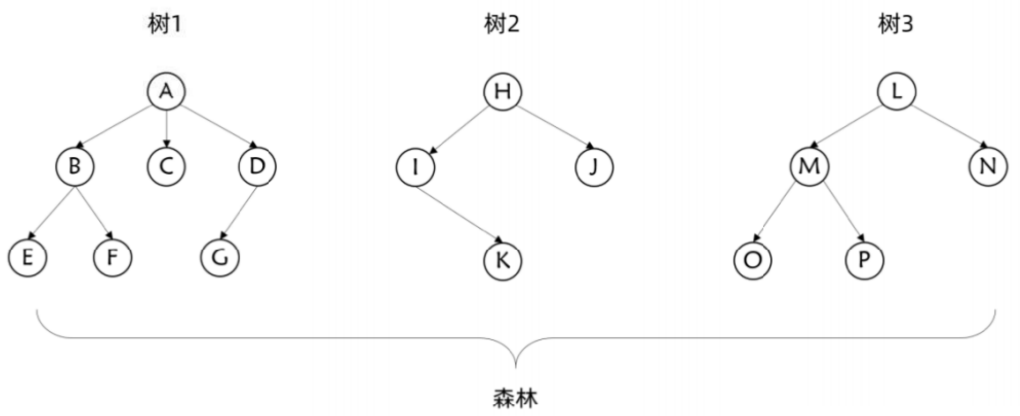
树的概念

除了根结点、叶子结点、树边等概念外，还有如下一些概念：

1、适用于无根树和有根树的概念

> *以下概念同时适用于无根树和有根树*

- **森林（forest）**：由 $m(m \geq 0)$ 棵互不相交的树构成的集合。按照定义，一棵树也是森林。
- **生成树（spanning tree）**：一个连通无向图的生成子图，同时要求是树。也即在有 n 个节点的图的边集中选择 $n-1$ 条，将所有顶点连通。
- **无根树的叶结点（leaf node）**：度数不超过 1 的结点。（为什么是“不超过 1”，而不是“恰为 1”？考虑树只有一个结点时，没有其他节点。）
- **有根树的叶结点（leaf node）**：没有子结点的结点。



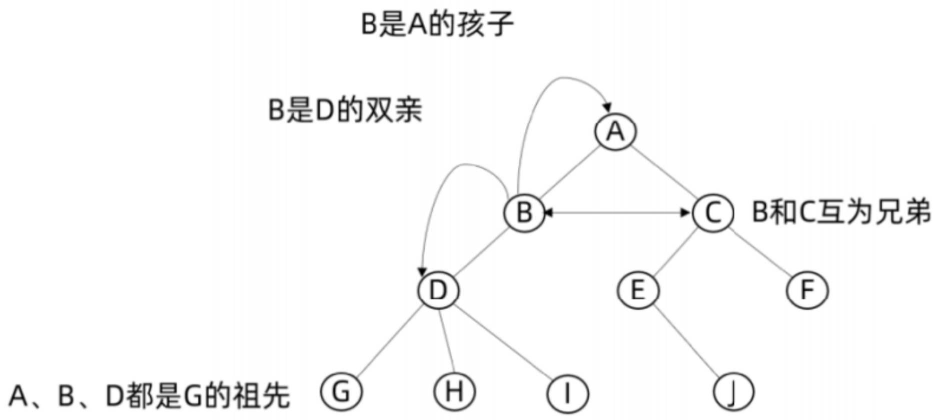
森林的概念

2、只适用于有根树的概念

> *以下概念只适用于有根树*

- **祖先（ancestor）**：一个结点到根结点的路径上，除了它本身外的结点构成的集合。根结点的祖先集合为空。
- **父结点（parent node）**：对于除根以外的每个结点，定义为从该结点到根路径上的第二个结点。❗❗注意：**根结点没有父结点**。（有的教材里也称之为父亲结点或双亲结点）
- **子结点（child node）**：如果 u 是 v 的父亲，那么 v 是 u 的子结点（孩子）。子结点的顺序一般不加以区分，二叉树是一个例外。
- **兄弟（sibling）**：同一个父亲的多个子结点互为兄弟。
- **堂兄弟**：同一层次的所有不互为兄弟的结点互为堂兄弟。
- **后代（descendant）**：子结点和子结点的后代。或者理解成：如果 u 是 v 的祖先，那么 v 是 u 的后代。
- **结点的度**：结点儿子的个数，称为结点的度。显然，叶子结点的度为 0。
- **结点的高度（depth）**：也称结点的层次或深度，指到根结点的路径上的边数。根节点的高度为 0，根的子结点高度为 1，以此类推。❗❗注意：也有一些场合为了方便会把根结点的高度定义为 1。
- **树的高度（height）**：也称树的深度，所有结点的深度的最大值。
- **树的度**：树中结点的最大度数称为树的度。
- **子树（subtree）**：删掉与父亲相连的边后，该结点所在的子图。

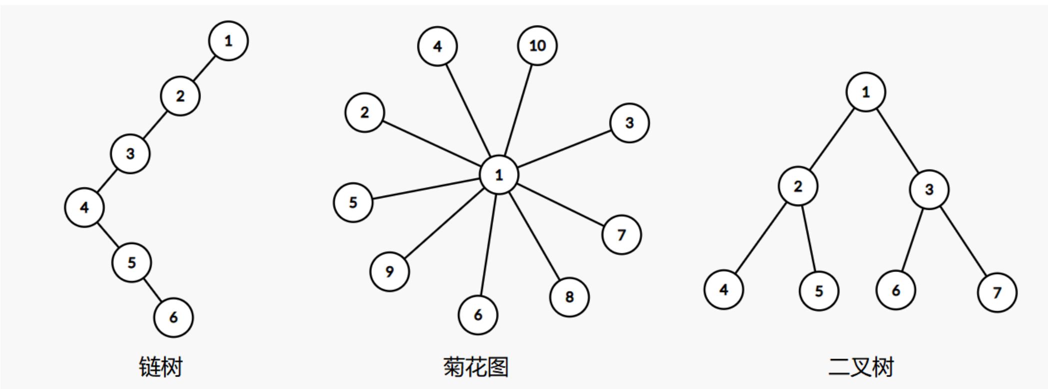
> *问：结点的深度和高度的区别？*
>
> *答：深度是从根结点开始**自顶向下**逐层累加的，高度是从叶结点开始**自底向上**逐层累加的。*



3、一些特殊的树

> *以下这些树很特殊，在思考和解决问题时，应注意算法和代码在以下特殊情况下会发生什么。有些时候，问题会考察这些特殊情况。*

- **链（chain/path graph）**：满足与任一结点相连的边不超过 2 条的树称为链。**一个有 n 个结点的树，当它是链树时，树的高度最大，达到 $n-1$。**
- **菊花/星星（star）**：满足存在 u 使得所有除 u 以外结点均与 u 相连的树称为菊花。**一个有 $n(n \geq 2)$ 个结点的树，当它是星树时，树的高度最小，为 1。**
- **有根二叉树（rooted binary tree）**：每个结点最多只有两个儿子（子结点）的有根树称为二叉树。常常对两个子结点的顺序加以区分，分别称之为左子结点和右子结点。大多数情况下，**二叉树** 一词均指有根二叉树。



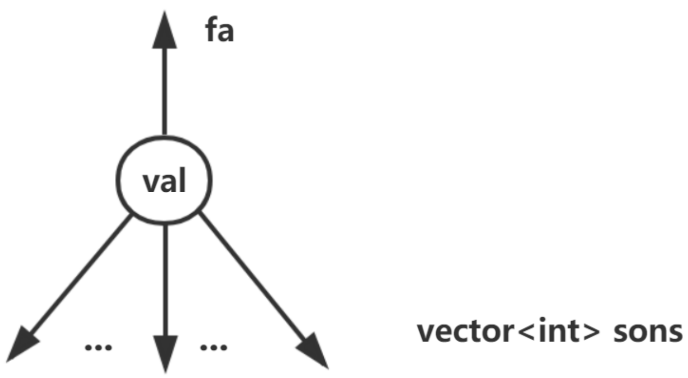
树的存储与遍历

引入

显然 **树** 的定义是递归的，**是一种递归的数据结构**。树作为一种逻辑结构，同时也是一种 **分层** 结构，具有以下两个特点：

1）树的根结点没有前驱结点，除根结点之外的所有结点有且只有一个前驱结点。

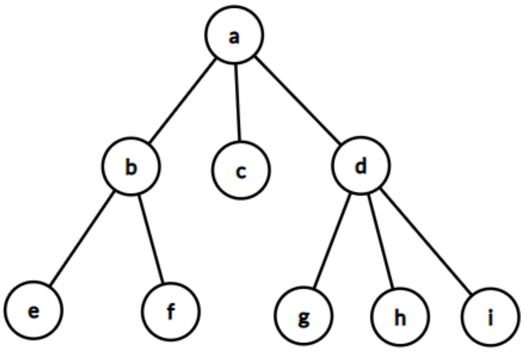
2）树中所有结点可以有零个或多个后继结点。



有根树的存储

父亲表示法

除根节点外，其他结点有且仅有一个父结点，因此，我们可以把每一条树边存储在其子结点上，形式为：i 结点的父亲是 j 结点，如下图所示。



下标	结点	fa[i]
0	-	0
1	a	0
2	b	1
3	c	1
4	d	1
5	e	2
6	f	2
7	g	4
8	h	4
9	i	4

树的父亲表示法存储示意图

父亲表示法的存储结构基本实现代码如下：

```
int fa[N];
fa[a] = 0;      // 表示 a 是根，也可用 -1 表示父结点不存在，具体取决于代码的写法
fa[x] = y;      // 连接 x 和 y，结点 x 的父结点是 y
```

如果需要保存结点信息，那么可以定义成结构体实现：

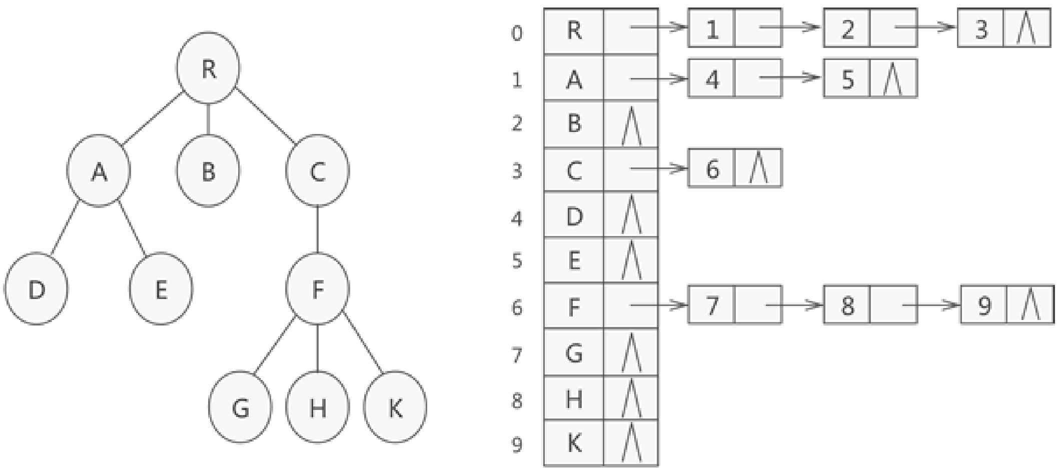
```
struct node{
    int val;              // 结点信息，也可能是其他数据类型
    int fa;               // 父结点(father)编号
};
node tree[N];             // 定义一棵树
void link(int x, int y)   // y 的父结点是 x
{
    tree[y].fa = x;       // 连接 x 和 y
}
```

优点：对于任意结点，可以方便找到其所有子结点；

缺点：找父结点很麻烦，需要遍历所有结点；

孩子表示法

对于每个结点，有一个数据域和多个指针域，数据域保存当前结点的数据，指针域的每个指针指向一个孩子结点 ，例如下图：



a) 普通树　　　　b) 孩子表示法

数组写法示例如下：

```
// 定义一棵树
int val[N];                // 结点信息，也可能是其他数据类型
vector<int> son[N];        // son[i]: i 的所有子结点编号


cin >> x >> y;      // x 是 y 的父结点
son[x].push_back(y);  // x 多了一个子节点 y
```

结构体写法示例代码如下：

```
struct node{
    int val;                // 结点信息，也可能是其他数据类型
    vector<int> son;        // son[i]: i 的所有子结点编号
};
node tree[N];                    // 定义一棵树

cin >> x >> y;    // x 是 y 的父结点
tree[x].son.push_back(y);  // x 多了一个子节点 y
```

优点：对于任意结点，可以方便找到其所有子结点；

缺点：找父结点很麻烦，需要遍历所有结点；

有根树的图存储方式

树其实是一种特殊的图，可以把一条树边看作一条父亲指向儿子（或儿子指向父亲）的有向边。因此图的存储方式也可用来存储树。图的存储有邻接矩阵、邻接表和链式前向星（特殊的邻接表）等实现方式。具体做法将在后面学习，这里只做简单说明。

邻接矩阵

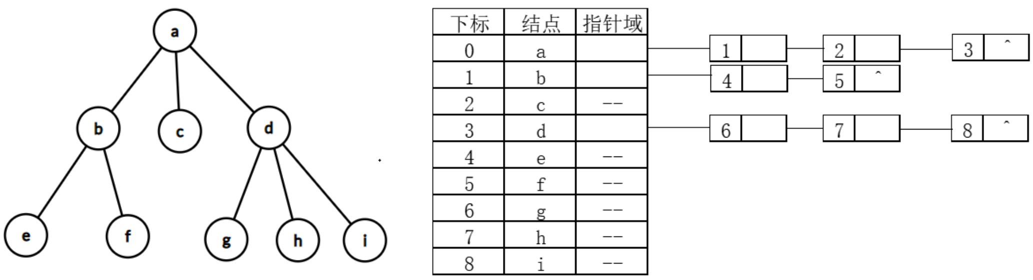
使用一个 $n \times n$ 的 bool 数组 mp，$mp[x][y]$ 为 true 则表示 x 到 y 存在有向边，反之则表示不能存在该有向边。

邻接矩阵存储代码如下：

```
bool mp[N][N];
void link(int x, int y)
{
    mp[x][y] = 1;    // x 到 y 存在有向边
}
```

邻接表

同样，我们可以采用邻接表来存储一个点连出的多条树边，如下图所示。



树的邻接表存储示意图

使用 C++ 的 STL 向量容器 vector 的邻接表存储代码如下：

```
vector<int> g[N];    // g[i] 存和结点 i 的出边

void link(int x, int y)  // 连一条由 x 指向 y 的边，在树中指 x 是 y 的父结点
{
    g[x].push_back(y);
}
```

树的遍历

定义

树的遍历是指 **从根节点出发**，按照 **某种次序** 依次访问树中的所有结点，使得 **每个结点都被访问仅被访问一次**。

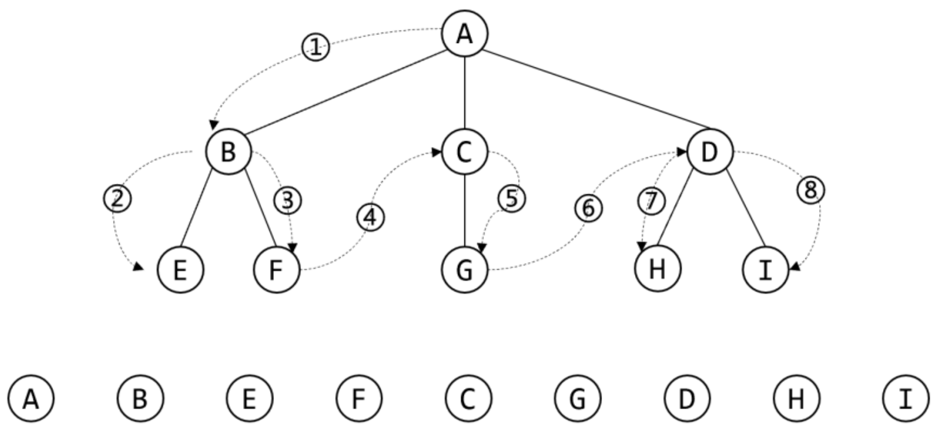
常用的遍历方式有：先根遍历（DFS）和层次遍历（BFS）。

先根遍历

遍历规则：

1. 若树为空，则停止遍历；
2. 如果树不为空，否则先访问根结点，再依次先根遍历根结点的所有子树。

可以看出，先根遍历是一种深度优先遍历。



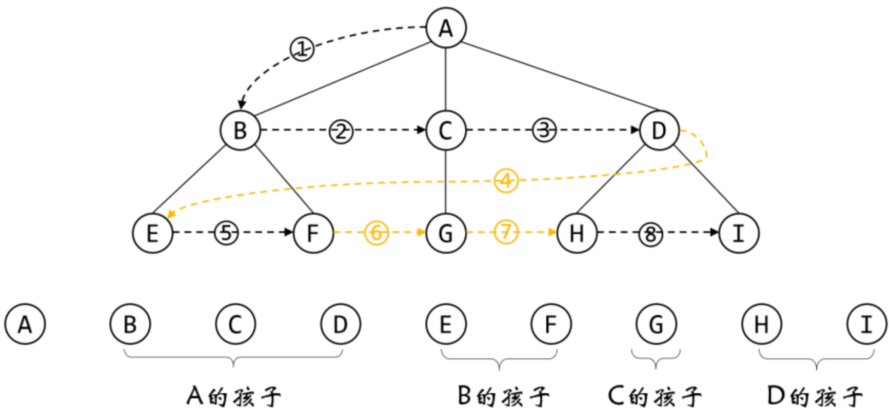
代码框架：

```
// 深度优先遍历树
void dfs(int root)
{
    // 1- 对当前结点进行处理，比如输出
    ......
    // 2- 扩展新结点，即遍历所有子节点
    for(int i = 0; i < tree[root].son.size(); i++)
    {
        dfs(tree[root].son[i]);  // 子节点入“函数栈”
    }
}
```

层次遍历

遍历规则为：

1. 若树为空，则停止遍历。
2. 如果树不为空，则从树的第一层，也就是根节点开始访问，从上而下逐层访问，在同一层中，按照从左到右的顺序对节点逐个访问。



如果某个结点先被访问，则其子节点也会先被访问，如果将访问本身看作入，将访问子节点看作出，则层次遍历的规则满足先进先出。

可以看出，层次遍历是一种广度优先遍历。

代码框架：

```
// 广度优先遍历
void bfs(int root)
```

```
{
    queue<int> q;         // 1- 定义队列
    q.push(root);         // 2- 根节点入队
    while(!q.empty())     // 3- 只要队列不空，就:
    {
        // (1) 取出队首结点，并对当前结点进行处理，比如输出
        int k = q.front();
        q.pop();
        ......
        // (2) 扩展新结点，即遍历所有子结点
        for(int i = 0; i < tree[k].son.size(); i++)
        {
            q.push(tree[k].son[i]);  // 子结点入队
        }
    }
}
```

树的深度优先遍历

```
#include<bits/stdc++.h>
using namespace std;

const int N = 105;
vector<int> t[N];   // 孩子表示法，存储树的结构

// 深度优先搜索函数
void dfs(int root)
{
    cout << root << ' '; // 输出当前节点编号
    if(t[root].empty()) return; // 如果当前节点没有孩子节点，则直接返回
    sort(t[root].begin(), t[root].end());   // 按编号从小到大遍历孩子节点
    for(int i = 0; i < t[root].size(); i++)
        dfs(t[root][i]); // 递归调用深度优先搜索函数，遍历孩子节点
}

int main()
```

```
{
    int n, x, y;
    cin >> n;
    n--; // 减去根节点的数量，因为根节点默认为1
    while(n--)
    {
        cin >> x >> y;
        t[x].push_back(y); // 构建树的结构，将孩子节点加入到对应父节点的孩子列表中
    }
    dfs(1); // 从根节点1开始深度优先搜索
    return 0;
}
```

树的广度优先遍历

```
#include<bits/stdc++.h>
using namespace std;

const int N = 105;
vector<int> t[N];     // 这题不需要去找父结点，所以用孩子表示法就足够了

void bfs(int root)
{
    queue<int> q;
    q.push(root);     // 根结点入队
    while(!q.empty())
    {
        // 1- 输出当前结点
        int k = q.front();  q.pop();
        cout << k << ' ';
        // 2- 扩展，从当前结点扩展新结点
        for(int i = 0; i < t[k].size(); i++)
            q.push(t[k][i]);
    }
}
```

```
int main()
{
    int n, x, y;
    cin >> n;
    n = n - 1;   // n-1 条边
    while(n--)
    {
        cin >> x >> y;
        t[x].push_back(y);
    }
    bfs(1);
    return 0;
}
```

树根和孩子

```
#include <bits/stdc++.h>
using namespace std;

// 节点结构体
struct Node {
    int parent;          // 存父节点编号
    vector<int> child;   // 存所有孩子
} a[101];

int n, m, x, y, maxn, maxd;

int main() {
    cin >> n >> m;

    // 读取边信息并构建树的结构
    for (int i = 1; i <= m; i++) {
        cin >> x >> y;
        a[y].parent = x;
        a[x].child.push_back(y);
    }

    // 找到根节点并统计最大孩子数
```

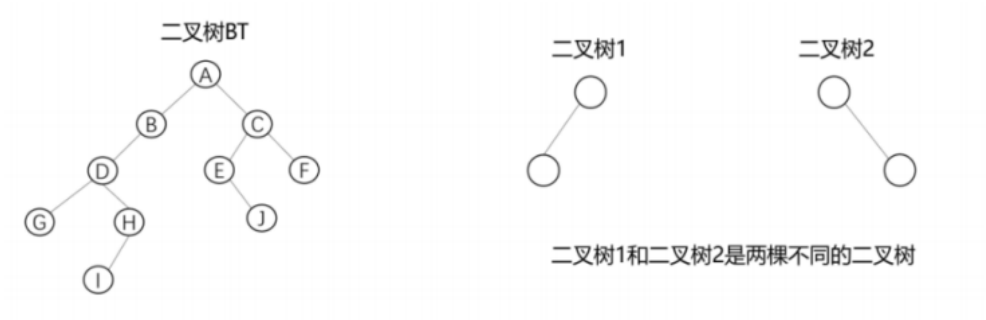
```
    for (int i = 1; i <= n; i++) {
        if (!a[i].parent)
            cout << i << endl; // 输出根节点编号
        if (a[i].child.size() >= maxn) {
            maxn = a[i].child.size();
            maxd = i;
        }
    }

    // 输出具有最大孩子数的节点及其孩子节点
    cout << maxd << endl; // 输出具有最大孩子数的节点编号
    sort(a[maxd].child.begin(), a[maxd].child.end()); // 对孩子节点进行排序
    for (int i = 0; i < a[maxd].child.size(); i++)
        cout << a[maxd].child[i] << " "; // 输出孩子节点
    return 0;
}
```

二叉树的概念

定义

简单地理解，满足以下两个条件的树就是二叉树：

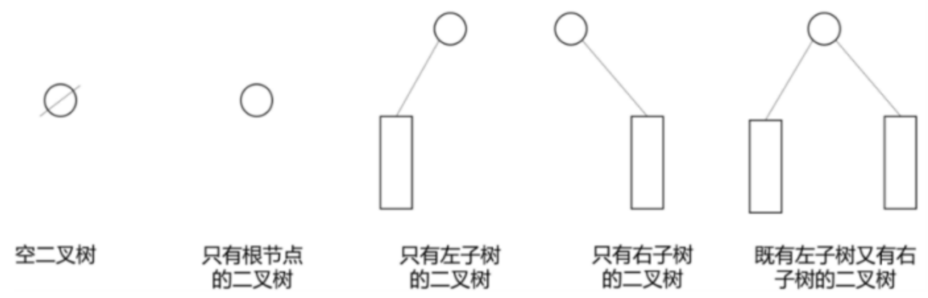
1. 树中包含的各个节点的度不能超过 2，即只能是 0、1 或者 2；
2. 本身是有序树，即左子树和右子树的顺序不能颠倒，即使只有一棵子树，也要区分是左子树还是右子树。



形态

注意：以下各种情况下，一棵树都是一棵二叉树！

1. 没有结点的树也是一棵树，称为空树！
2. 只有一个根结点也是一棵树；
3. 只有左子树的二叉树也是二叉树；
4. 只有右子树的二叉树也是二叉树；
5. 既有左子树，又有右子树的二叉树是一棵二叉树；



几种特殊的二叉树

1 斜树

顾名思义，斜树一定是斜的，但是往哪边斜还是有讲究的。

① 所有节点都只有左子树的二叉树称为 **左斜树**。

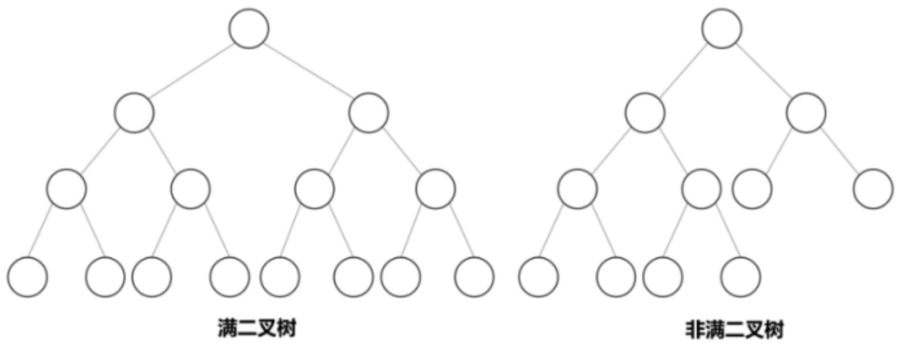
② 所有节点都只有右子树的二叉树称为 **右斜树**。

③ 斜树的每一层都只有一个节点，节点的个数和二叉树的深度相同。

2 满二叉树

在一棵二叉树中，如果所有分支节点都存在左子树和右子树（度为 2），并且所有叶子节点都在同一层上，这样

的二叉树称为满二叉树。

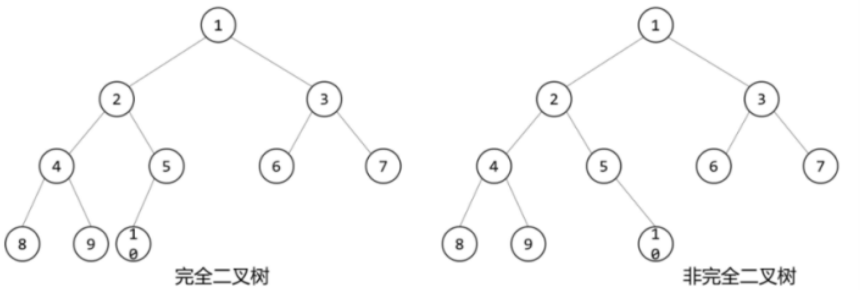


3 完全二叉树

对一棵具有 n 个节点的二叉树按层次序编号，如果编号为 $i(1 < i \leq n)$ 的节点与同样深度的满二叉树中编号

为 i 的节点在二叉树中位置完全相同，则这棵二叉树称为完全二叉树。

也可以这么去定义完全二叉树：如果二叉树中除了最下层，其他每层都饱满，最下层的结点都集中在该层最左边的若干位置上，则此二叉树被称为完全二叉树。



注意：

（1）满二叉树是完全二叉树，完全二叉树不一定是满二叉树！

（2）在满二叉树的最下层上，从最右边开始连续删去若干结点后得到的二叉树仍然是一棵完全二叉树。

（3）在完全二叉树中，若某个结点没有左孩子，则它一定没有右孩子，即该结点必是叶结点。

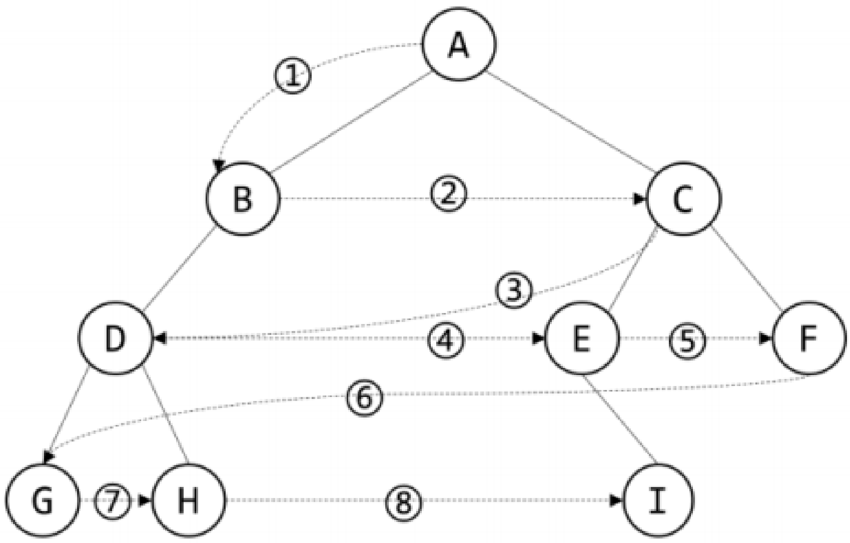
二叉树的遍历

树的遍历是指 **从根节点出发**，按照 **某种次序** 依次访问树中的所有结点，使得 **每个结点都被访问仅被访问一次**。

层次遍历

即广度优先遍历，遍历规则为：

1. 若二叉树为空，则停止遍历并返回。
2. 如果树不为空，则从根节点开始访问，从上而下逐层访问，在同一层中，按照左孩子、右孩子的顺序对节点逐个访问。



示例代码：

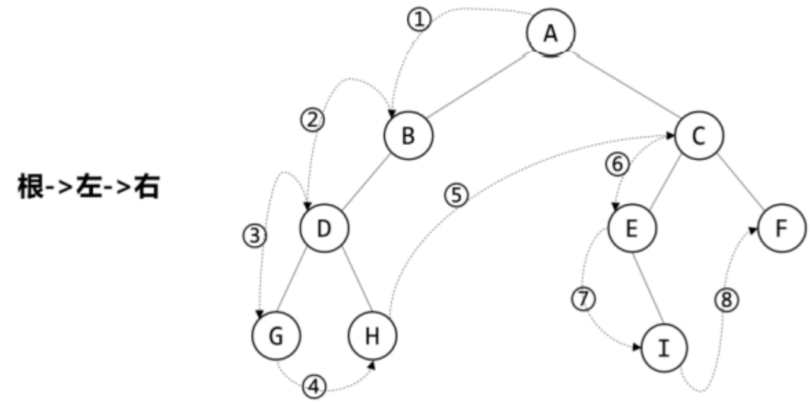
```
const int N  = 10005;
char val[N];                // 存当前结点的信息
int l[N], r[N];    //  left[i],right[i]: 结点 i 的左孩子和右孩子编
号
void bfs(int root)
{
    queue<int> q;
    q.push(root);
```

```
    while(!q.empty())
    {
        int k = q.front(); q.pop();
        cout << k << ' ';
        if(l[k]) q.push(l[k]);
        if(r[k]) q.push(r[k]);
    }
}
```

先序遍历

又称先根遍历、前序遍历。对于一棵非空的树，是指这样的一种遍历顺序：

1. 访问根节点；
2. 如果左子树不是空的，那么按照先序遍历的顺序遍历左子树；
3. 如果右子树不是空的，那么按照先序遍历的顺序遍历右子树；



先序遍历可以总结成三个字： **根左右**。容易看出，这种遍历方式的定义是一种递归定义的方式。因此，在代码实现上，先序遍历也是采用递归来实现的。示例代码如下：

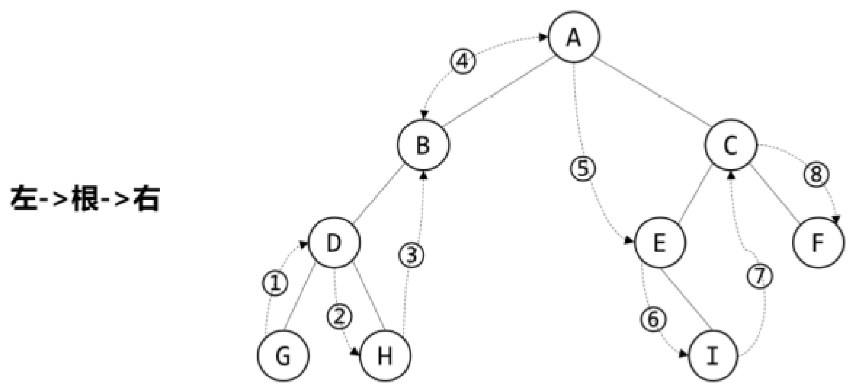
```
// 结构体存树
struct node{
    char val;
    int left, right;
} t[N];

void visit(int node)
```

```
{
    cout << t[root].val;    // 输出当前结点—子树的根结点
}

void pre_order(int root)
{
    visit(root);                 // 访问根节点
    if(t[root].left)
        pre_order(t[root].left);  // 左子树不空，遍历左子树
    if(t[root].right)
        pre_order(t[root].right); // 右子树不空，遍历右子树
}
```

中序遍历

又称中根遍历。对于一棵非空的树，是指这样的一种遍历顺序：

1. 如果根节点的左子树不是空的，那么按照中序遍历的顺序遍历左子树；
2. 访问根节点；
3. 如果根节点的右子树不是空的，那么按照中序遍历的顺序遍历右子树；



中序遍历可以总结成三个字：**左根右**。容易看出，它和先序遍历一样都是递归定义，且只是遍历的顺序不同。示例代码如下：

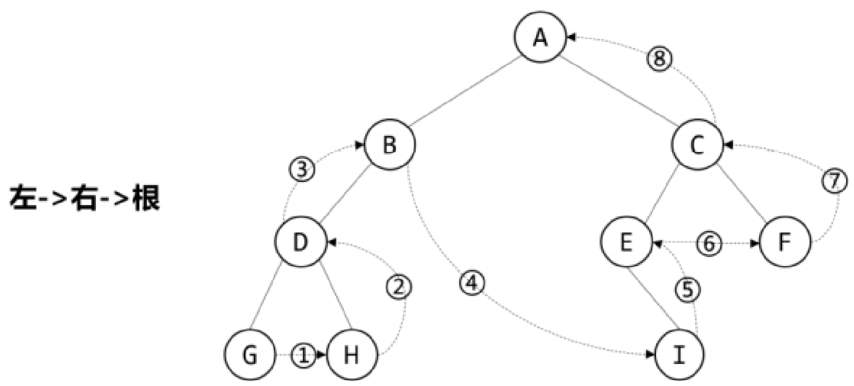
```
// 结构体存树
struct node{
    char val;
```

```
    int left, right;
} t[N];

void visit(int node)
{
    cout << t[root].val;    // 输出当前结点——子树的根结点
}

void in_order(int root)
{
    if(t[root].left)
        in_order(t[root].left);  // 左子树不空，遍历左子树
    visit(root);                  // 访问根节点
    if(t[root].right)
        in_order(t[root].right); // 右子树不空，遍历右子树
}
```

后序遍历

又称后根遍历。对于一棵非空的树，是指这样的一种遍历顺序：

1. 如果根节点的左子树不是空的，那么按照后序遍历的顺序遍历左子树；
2. 如果根节点的右子树不是空的，那么按照后序遍历的顺序遍历右子树；
3. 访问根节点；



后序遍历可以总结成三个字： **左右根**。容易看出，它和先序遍历一样都是递归定义，且只是遍历的顺序不同。示例代码如下：

```
// 结构体存树
struct node{
    char val;
    int left, right;
} t[N];

void visit(int node)
{
    cout << t[root].val;    // 输出当前结点—子树的根结点
}

void in_order(int root)
{
    if(t[root].left)
        in_order(t[root].left);  // 左子树不空，遍历左子树
    if(t[root].right)
        in_order(t[root].right); // 右子树不空，遍历右子树
    visit(root);                  // 访问根节点
}
```

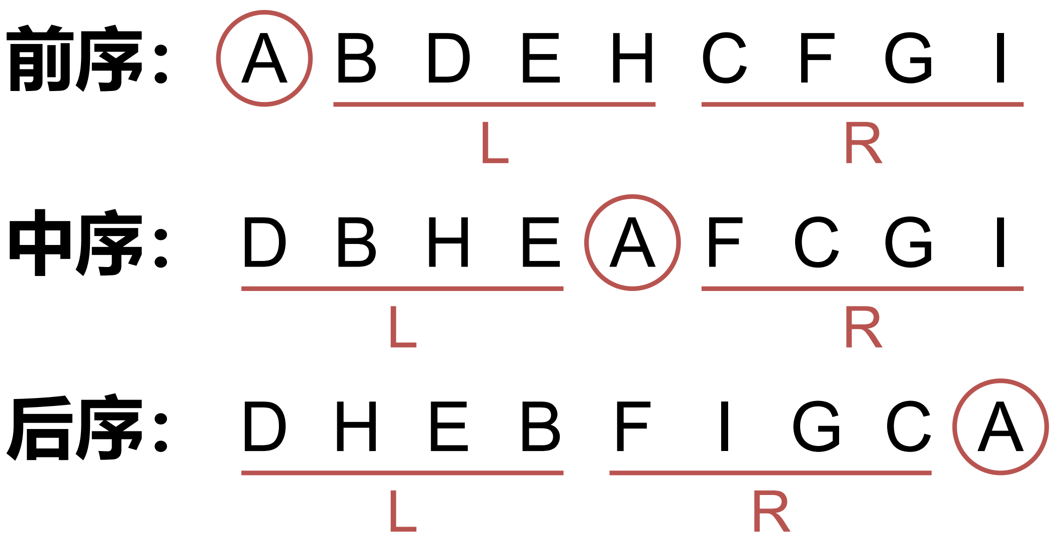
二叉树的三种遍历

```
#include <bits/stdc++.h>
using namespace std;

// 存节点信息
struct node{
    char v;
    int l, r;  // 左节点、右节点编号
} t[30];

char c;
int n, l, r, x;  // x:根节点编号
int in[30];         // in[i]: 节点 i 的入度，用来寻找根节点

// 前(先)序遍历：根左右
void dfs1(int x) {
    cout << t[x].v;
```

```
    if(t[x].l) dfs1(t[x].l);
    if(t[x].r) dfs1(t[x].r);
}
// 中序遍历：左根右
void dfs2(int x) {
    if(t[x].l) dfs2(t[x].l);
    cout << t[x].v;
    if(t[x].r) dfs2(t[x].r);
}
// 后序遍历：左右根
void dfs3(int x) {
    if(t[x].l) dfs3(t[x].l);
    if(t[x].r) dfs3(t[x].r);
    cout << t[x].v;
}

int main() {
    cin >> n;
    for(int i = 1; i <= n; i++) { // 读入并存储节点信息
        cin >> c >> l >> r;
        t[i] = (node){c,l,r};
        in[l]++; in[r]++;
    }
    // 注意要先找根
    for(int i = 1; i <= n; i++) { // 依据入度判断根节点
        if(in[i] == 0) {  // 根节点入度为零
            x = i;
            break;
        }
    }
    // 进行三种遍历
    dfs1(x);
    cout << '\n';
    dfs2(x);
    cout << '\n';
    dfs3(x);
    cout << '\n';
    return 0;
}
```

已知二叉树中序后序，求前序

（下面图片来自OI-Wiki）




```cpp
#include<bits/stdc++.h>
using namespace std;

struct node{
    char val;
    int l=-1, r=-1;
}tr[30];
string sa, sb;

int build(int la, int lb, int ra, int rb)
{
    if(la == lb) return la;    // 叶子节点
    if(la > lb) return -1;     // 空子树
    int pr = la;    //
    while(pr <= lb && sa[pr] != sb[rb]) pr++;  // 中序中找树根
    tr[pr].l = build(la, pr-1, ra, ra+(pr-la-1));  // 建左子树
    tr[pr].r = build(pr+1, lb, ra+pr-la, rb-1);    // 建右子树
    return pr;
}

void pre_order(int root)
```

```
{
    cout << sa[root];
    if(tr[root].l!=-1) pre_order(tr[root].l);
    if(tr[root].r!=-1) pre_order(tr[root].r);
}
int main()
{
    cin>>sa>>sb;
    int r = sa.size()-1;
    int root = build(0,r,0,r);
    pre_order(root);
    return 0;
}
```

练习题

完全二叉树的权值（蓝桥杯C/C++2019B组省赛）

二叉树的秘密

最近公共祖先（LCA）（了解）

定义

我们之前已经了解到“祖先”的概念，若两个节点的祖先相同，则叫做两个节点的 **公共祖先**。

最近公共祖先（Lowest Common Ancestor），简称 **LCA**，两个节点的最近公共祖先，就是这两个点的公共祖先里面，距离两个节点最近的公共祖先（离根节点最远）。

求法

方法一、朴素算法（暴力跳）

可以每次找深度比较大的那个点，让它向上跳。显然在树上，这两个点最后一定会相遇，相遇的位置就是想要求的 LCA。

或者先向上调整深度较大的点，令他们深度相同，然后再共同向上跳转，最后也一定会相遇。

朴素算法预处理时需要DFS整棵树，时间复杂度为 $O(n)$，**单次查询** 时间复杂度为 $O(n)$。但由于随机树高为 $O(\log n)$，所以朴素算法在随机树上的单次查询时间复杂度为 $O(\log n)$。

参考代码：

```
#include <bits/stdc++.h>
using namespace std;
const int N = 5e5+5;
int n, m, s, x, y, fa[N], d[N];
int h[N], e[2*N], nxt[2*N], cnt;
void add(int x, int y)
```

```
{
    nxt[++cnt] = h[x];
    e[cnt] = y;
    h[x] = cnt;
}

// 求每个节点的深度和 fa
void dfs(int k, int f)
{
    fa[k] = f;  d[k] = d[f]+1;
    for(int i = h[k]; i; i=nxt[i])
    {
        if(e[i] != f) dfs(e[i], k);
    }
}

// 方法一、每次让深度更大的节点往上跳
int lca(int x, int y)
{
    while(x!=y) {
        if(d[x] > d[y]) x = fa[x];
        else y = fa[y];
    }
    return x;
}

// 方法二、先向上调整深度较大的点，令他们深度相同，然后再共同向上跳转
int lca(int x, int y)
{
    // 把 x 移动到和 y 同一高度
    if(d[x] < d[y]) swap(x, y);
    while(d[x] > d[y]) x=fa[x];
    // 开始同步往上跳
    while(x!=y) {
        x = fa[x];
        y = fa[y];
    }
    return x;
}
```

```
int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);
    cin >> n >> m >> s;
    for(int i = 1; i < n; i++)
    {
        cin >> x >> y;
        add(x, y); add(y, x);
    }
    // dfs 预处理获取深度和 fa 数组
    dfs(s, 1);
    for(int i = 1; i <= m; i++)
    {
        cin >> x >> y;
        cout << lca(x, y) << '\n';
    }
    return 0;
}
```

采用朴素算法求 LCA 的时间复杂度与两点间的距离有关，极限情况可达到 $O(n)$。虽然该算法时间复杂度较高，但该算法也是有一定的应用价值的

方法二、倍增算法

本算法是对算法一中一步步走的改进，核心实质是让两个结点 **每次向上走 2 的幂次步**，具体操作如下：

第一步，求出倍增数组： 首先开一个 $n \times \log n$ 的数组，比如 $fa[n][\log n]$，其中 $fa[i][j]$ 表示 i **节点的第** 2^j **个父亲**。$fa[i][j]$ 为结点 i 向上 走 2^j 步后能走到的结点。

我们规定根结点的父亲是它自己，这样根结点往上走还是在根结点。

对于 $j = 0$，$fa[i][j]$ 就是结点 i 的父亲。

对于 $j > 0, fa[i][j]$ 等于 $fa[fa[i][j-1]][j-1]$（即结点 i 往上走 2^{j-1} 步后再往上走 2^{j-1} 步)。

我们通过一遍从根节点开始的 `dfs` 预处理出 fa 数组和深度数组 d（可以方便地一起处理出来，后面要用到）。

第二步，把两个点移到同一深度： 假设要求 $LCA(x，y)$，不失一般性，令 $d[x] \geq d[y]$（x 的深度大于等于 y 的深度，否则我们交换 x, y），先让 x 往上走 $d[x]-d[y]$ 步。我们对这个差进行二进制拆分，就可以通过倍增数组往上走 2 的幂次步（即对于二进制为 1 的第 i 位。要往上走 2^i 步，即调用 $x = fa[x][i]$），那么可以在 $O(\log_2 n)$ 的时间复杂度内到达目标深度。或者说，类似的，我们也可以从大往小扫描 i，一直尝试到 0（包括 0），如果每次 $fa[x][i]$ 深度不小于 y，我们就跳 x。两种做法效果是一样的，读者可以根据自己的喜好选择。

第三步，求出 LCA： 假设 x 与 y 向上走最小的 L 步后是同一结点，也就意味着，x 与 y 向上走最大的 $L-1$ 步，也是不同的结点。我们可以从大到小枚举往上走 2^i 步，如果当前 x 与 y 向上走 2^i 步后为同一点，则停止，否则一起往上走。这样，我们就能在 $\log n$ 的时间复杂度内使 x 与 y 都向上走 $L-1$ 步。根据倍增数组是 2 的幂次这个特性，这样的做法可以看成一个通过 **二分法** 来求解走最大 $L-1$ 步的过程。用这样的方法从大到小决策 $logn$ 次，直到完全不能向上走了为止，我们再让 x 与 y 各向上走一步，则为 $LCA(x, y)$。

倍增算法的预处理时间复杂度为 $O(n \log n)$， **单次查询** 时间复杂度为 $O(\log n)$。

另外倍增算法可以通过交换 `fa` 数组的两维使较小维放在前面。这样可以减少 cache miss 次数，提高程序效率。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 5e5+5;
int n, m, s, x, y, fa[N][20], d[N], logn[N];
int h[N], e[2*N], nxt[2*N], cnt;
void add(int x, int y)
{
    nxt[++cnt] = h[x];
    e[cnt] = y;
    h[x] = cnt;
}

// 求每个节点的深度和 fa
void dfs(int u, int f)
```

```
{
    fa[u][0] = f;  d[u] = d[f]+1;
    // 二分法计算 u 的第 2^i 个父结点
    // 是 u 的第 2^{i-1} 个祖先的第 2^{i-1} 个祖先
    for(int i = 1; i <= logn[d[u]]; i++)
        fa[u][i] = fa[fa[u][i-1]][i-1];
    for(int i = h[u]; i; i=nxt[i])
    {
        if(e[i] != f) dfs(e[i], u);
    }
}

// 先向上调整深度较大的点，令他们深度相同，然后再共同向上跳转
int lca(int x, int y)
{
    // 把 x 移动到和 y 同一高度
    if(d[x] < d[y]) swap(x, y);
    while(d[x] > d[y]) x=fa[x][logn[d[x]-d[y]]];
    if(x == y) return x;
    // 开始同步往上跳 2^i 步
    for(int i = logn[d[x]]; ~i; i--) {
        // 只要往上跳 2^i 步父节点还不相同，就继续上跳
        if(fa[x][i] != fa[y][i])
            x = fa[x][i], y = fa[y][i];
    }
    return fa[x][0];   // 第一个父结点就是 LCA
}

int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);
    cin >> n >> m >> s;
    for(int i = 1; i < n; i++)
    {
        cin >> x >> y;
        add(x, y); add(y, x);
    }
    // 预处理 log_2 数组 logn[i]
```

```
    for(int i = 2; i <= n; i++)
        logn[i] = logn[i/2] + 1;
    // dfs 预处理获取深度和 fa 数组
    dfs(s, 0);
    for(int i = 1; i <= m; i++)
    {
        cin >> x >> y;
        cout << lca(x, y) << '\n';
    }
    return 0;
}
```

景区导游（蓝桥杯C/C++2023B组省赛）

```
#include <bits/stdc++.h>
#define int long long
using namespace std;

const int N = 1e5 + 10, M = 2 * N;
int h[N], e[M], ne[M], w[M], idx;
int depth[N], fa[N][21]; // 用于存储节点深度和节点的祖先信息
int aa[N], dist[N]; // 存储每个点到根节点的距离和每个点所在的位置
int n, m; // 点的数量和查询的数量

// 添加边函数
void add(int a, int b, int c) {
    e[++idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx;
}

// BFS计算每个节点的深度和距离
void bfs() {
    memset(depth, 0x3f, sizeof depth); // 初始化深度数组
    depth[0] = 0, depth[1] = 1; // 根节点深度为1
    queue<int> q;
    q.push(1); // 将根节点加入队列

    while (q.size()) { // 队列非空时循环
        int t = q.front();
```

```
        q.pop();

        for (int i = h[t]; i; i = ne[i]) { // 遍历节点 t 的邻接
节点
            int j = e[i];
            if (depth[j] > depth[t] + 1) { // 如果当前节点的深度
比之前的节点深度要小，则更新深度和距离
                depth[j] = depth[t] + 1;
                dist[j] = dist[t] + w[i];
                q.push(j);
                fa[j][0] = t; // 记录节点 j 的父节点

                // 更新节点的祖先信息
                for (int k = 1; (1 << k) <= depth[j]; k++)
                    fa[j][k] = fa[fa[j][k - 1]][k - 1];
            }
        }
    }
}

// 求两个点的最近公共祖先
int lca(int a, int b) {
    if (depth[a] < depth[b]) swap(a, b); // 保证 a 的深度不小于
b 的深度
    for (int k = 20; k >= 0; k--) { // 从深度高的节点开始往上跳，
直到两个节点在同一深度
        if (depth[fa[a][k]] >= depth[b])
            a = fa[a][k];
    }

    if (a == b) return a; // 如果 a 和 b 相等，说明其中一个是另一
个的祖先

    // 一起往上跳，直到找到最近公共祖先
    for (int k = 20; k >= 0; k--) {
        if (fa[a][k] != fa[b][k]) {
            a = fa[a][k];
            b = fa[b][k];
        }
```

```
    }

    return fa[a][0]; // 返回最近公共祖先节点
}

// 计算两个点的距离
int get_dist(int a, int b) {
    return dist[aa[a]] + dist[aa[b]] - 2 * dist[lca(aa[a],
aa[b])];
}

signed main() {
    cin >> n >> m; // 读入节点数量和查询次数
    memset(h, -1, sizeof h); // 初始化头节点数组

    // 读入边的信息
    for (int i = 1; i <= n - 1; i++) {
        int a, b, c;
        cin >> a >> b >> c;
        add(a, b, c), add(b, a, c); // 添加无向边
    }

    bfs(); // 进行BFS，计算深度和距离

    int sum = 0;
    for (int i = 1; i <= m; i++) {
        cin >> aa[i]; // 读入查询节点的位置
        if (i > 1) sum += get_dist(i, i - 1); // 计算相邻节点的
距离和
    }

    // 输出结果
    for (int i = 1; i <= m; i++) {
        if (i == 1) cout << sum - get_dist(i, i + 1) << " ";
        else if (i == m) cout << sum - get_dist(i - 1, i) <<
endl;
        else cout << sum - get_dist(i - 1, i) - get_dist(i, i +
1) + get_dist(i - 1, i + 1) << " ";
    }
```

```
    return 0;
}
```

练习题

砍树（蓝桥杯C/C++2023年B组省赛）